Towards Constructive Homological Algebra in Type Theory

Thierry Coquand, Arnaud Spiwack

Towards Constructive Homological Algebra in Type Theory

Thierry Coquand and Arnaud Spiwack

[1] Göteborg University
[2] Ecole Normale Supérieure de Cachan

Abstract. This paper reports on ongoing work on the project of representing the Kenzo system [15] in type theory [11].

Introduction

This paper reports on ongoing work on the project of representing the system Kenzo [15] in type theory. Kenzo is a symbolic computation system in algebraic topology, based on a rich mathematical theory described in [15], and which uses in an essential way ideas from functional programming. Our ultimate goal is to represent the mathematical results of [15] as constructive mathematical results developed in type theory. Using type theory as a functional programming language, this representation should then give a fully specified and checked functional version of Kenzo. Besides the Kenzo system, we can hope to develop in this way a library of reasonably efficient algorithms in homological algebra, similar to [5] but specified and written in type theory.

One of the main mathematical results on which the system Kenzo relies (the Basic Pertubation Lemma) has been already checked in the system Isabelle [1, 15]. Our paper complements this work by exploring the formalisation of Kenzo at the level of *preabelian category*. For this, we show in detail how to represent category theory, and in particular preabelian categories, in type theory. We use then this formalisation on the test example suggested in [1], where it is explained why it is quite subtle to represent it formally. We believe that to express reasoning at the level of category theory, in a characteristic "pointfree" way, is perfect for formalisation. Indeed it works well on this test example, and the formal reasoning in type theory follows closely the informal argument. We can then instantiate this abstract argument on the example of the category of abelian groups to get back the statement in [1].

This paper is organised as follows. First we describe in detail the general setting in which we represent the mathematics of Kenzo: dependent type theory with universes, essentially the system [11], with a special universe of propositions. The system Coq [6] is a possible implementation of this system. We explain then how usual mathematical notions (sets, groups, ...) and then the notion of category theory are represented in this setting. The main example is the notion of preabelian category. We show how a test example [1] can be represented as a general property of preabelian categories. This has been done formally in Coq. We end by listing the remaining steps for having a representation of Kenzo. An appendix presents the formal statements, that are reasonably close to the informal statements.

The results of this paper are quite preliminary w.r.t. to the general goal of actually running Kenzo program in type theory. They show however that this project should be feasible, and provide already interesting observations on the formal representation of mathematics in type theory.

1 Type theory

We shall use type theory [11] as a model for the mathematics used in homological algebra. It is an alternative to the system ZF, with closer connection with functional programming (and should be thus a priori well adapted for representing Kenzo). All mathematical notions and proofs are represented as λ-terms. These terms can then be directly computed [10] and there exist now actual efficient implementations of such computations [8].

The terms are untyped lambda terms with constants. We consider terms up to α-conversion. We have a constructor Π of arity 2 and we write $\Pi x{:}A.B$ instead of $\Pi\ A\ (\lambda x.B)$, and $A \to B$ instead of $\Pi\ A\ (\lambda x.B)$ if x is not free in B. We write also $\Pi x_1\ \ldots\ x_n : A.B$ for $\Pi x_1 : A. \ldots . \Pi x_n : A.B$. We write $B[x = M]$, or even $B[M]$ if x is clear, the substitution of the term M for the variable x in B. We have special constants $\mathsf{U}_1, \mathsf{U}_2, \ldots$ for universes. A *context* is a sequence $x_1 : A_1, \ldots, x_n : A_n$.

The minimal type theory we use has three forms of judgements

$$\Gamma \vdash A \qquad \Gamma \vdash M : A \qquad \Gamma \vdash$$

The last judgement $\Gamma \vdash$ expresses that Γ is a well-typed context. We may write $J\ [x : A]$ for $x : A \vdash J$.

The typing rules are as follows.

$$\frac{}{\vdash} \qquad \frac{\Gamma \vdash A}{\Gamma, x : A \vdash}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{U}_i} \qquad \frac{\Gamma \vdash A : \mathsf{U}_i}{\Gamma \vdash A} \qquad \frac{\Gamma, x : A \vdash B}{\Gamma \vdash \Pi x{:}A.B}$$

$$\frac{(x : A) \in \Gamma \quad \Gamma \vdash}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : \Pi x{:}A.B} \qquad \frac{\Gamma \vdash N : \Pi x{:}A.B \quad \Gamma \vdash M : A}{\Gamma \vdash N\ M : B[M]}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B \quad A =_\beta B}{\Gamma \vdash M : B}$$

We have also

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{U}_i : \mathsf{U}_{i+1}} \qquad \frac{\Gamma \vdash A : \mathsf{U}_i}{\Gamma \vdash A : \mathsf{U}_{i+1}}$$

We express finally that each universe U_i is closed under the product operation.

$$\frac{\Gamma \vdash A : \mathsf{U}_i \quad \Gamma, x : A \vdash B : \mathsf{U}_i}{\Gamma \vdash \Pi x{:}A.B : \mathsf{U}_i}$$

These rules have an intuitive interpretation in set theory, where U_i are Grothendieck universes [2]. The dependent product $\Pi x : A.B$ if $B(x)$ is a family of sets over a set A is the set of all families $(b_x)_{x \in A}$ such that $b_x \in B(x)$ for all $x \in A$.

It is convenient to introduce sigma types, with the following rules, and adding the conversion rules $(M_1, M_2).1 =_\beta M_1,\ (M_1, M_2).2 =_\beta M_2$

$$\frac{\Gamma, x : A \vdash B}{\Gamma \vdash \Sigma x : A.B}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B[M]}{\Gamma \vdash (M, N) : \Sigma x : A.B} \qquad \frac{\Gamma \vdash P : \Sigma x : A.B}{\Gamma \vdash P.1 : A} \qquad \frac{\Gamma \vdash P : \Sigma x : A.B}{\Gamma \vdash P.2 : B[P.1]}$$

$$\frac{\Gamma \vdash A : \mathsf{U}_i \quad \Gamma, x : A \vdash B : \mathsf{U}_i}{\Gamma \vdash \Sigma x : A.B : \mathsf{U}_i}$$

In set theory, $\Sigma x : A.B$ is the set of pairs (x, b_x) with $x \in A$ and $b_x \in B(x)$. We shall write $(x_1, \ldots, x_n)$ for $(\ldots(x_1, x_2), \ldots, x_n)$[3].

[3] The addition of sigma types is convenient but not strictly necessary. One can work with *vectors* of terms and *telescopes* instead [4]. A telescope is like a context $T = x_1 : A_1, \ldots, x_{n-1} : A_{n-1}, A_n$ and a vector $P_1, \ldots, P_n$ fits this telescope iff $P_1 : A_1, \ldots, P_n : A_n[P_1, \ldots, P_{n-1}]$.

2 A type of propositions

It is convenient also to introduce a special universe U_0, which in set theory would be the set of truth values, with the special rules

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{U}_0 : \mathsf{U}_1} \qquad \frac{\Gamma \vdash A : \mathsf{U}_0}{\Gamma \vdash A : \mathsf{U}_1} \qquad \frac{\Gamma \vdash A \quad \Gamma, x : A \vdash B : \mathsf{U}_0}{\Gamma \vdash \forall x : A.B : \mathsf{U}_0}$$

$$\frac{\Gamma, x : A \vdash B : \mathsf{U}_0 \quad \Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : \ \forall x : A.B} \qquad \frac{\Gamma \vdash N : \forall x : A.B \quad \Gamma \vdash M : A}{\Gamma \vdash N \ M : B[M]}$$

We write $A \Rightarrow B$ for $\forall x : A.B$ if x is not free in B, and $A_1 \Rightarrow \ldots \Rightarrow A_n$ denotes $(\ldots(A_1 \Rightarrow A_2) \ldots \Rightarrow A_n$.

We can quantify over any type: $\forall x : A.B : \mathsf{U}_0$ for any type A, if $B : \mathsf{U}_0$ $[x : A]$. In particular, we have $\forall x : \mathsf{U}_0.B : \mathsf{U}_0$ if $B : \mathsf{U}_0$ $[x : \mathsf{U}_0]$. This impredicativity is convenient, but not necessary for representing the reasonings in [15]. We can represent logical connectives as operations of type $\mathsf{U}_0 \to \mathsf{U}_0 \to \mathsf{U}_0$. For instance $A \wedge B$ is defined as

$$\forall P : \mathsf{U}_0.(A \Rightarrow B \Rightarrow P) \Rightarrow P$$

We define also $A \Leftrightarrow B$ as $(A \Rightarrow B) \wedge (B \Rightarrow A)$ and $\bot : \mathsf{U}_0$ as $\forall A : \mathsf{U}_0.A$ and $\top : \mathsf{U}_0$ as $\forall A : \mathsf{U}_0.A \Rightarrow A$, which has an inhabitant $\lambda A \lambda x.x$.

These rules have also a direct interpretation in ZF set theory. The type U_0 is interpreted as the set $\{0, 1\}$, 0 being the empty set and 1 being the set $\{0\}$, and $\forall x : A.B$ is 1 iff $B(x) = 1$ for all x in A and is 0 otherwise. See [13] for a careful presentation of this model.

Notice that if $A : \mathsf{U}_1$ and $B : \mathsf{U}_0$ $[x : A]$ we have also $B : \mathsf{U}_1$ $[x : A]$ and we can also form $\Pi x : A.B : \mathsf{U}_1$. There are two maps

$$(\Pi x : A.B) \to \forall x : A.B \qquad (\forall x : A.B) \to \Pi x : A.B$$

but the two types $\forall x : A.B$ and $\Pi x : A.B$ are not convertible. (They are not identical in general in the set theoretical model. Indeed, in this model an element of $\Pi x : A.B$ will be a family, which in set theory has to be the set of pairs $(a, 0)$ with a in A, while an element of $\forall x : A.B$ can only be the empty set 0. See [13] for a general discussion of this point.)

The existence of a set theoretical model entails the *consistency* of our type theory: the type $\bot$ is not inhabited (*i.e.* there is no proof of false). A sharper result is the strong normalisation theorem [12]. It follows from this that if M, A are in normal form then the judgement $\vdash M : A$ is decidable. It follows from this that one can actually checked the correctness of proofs. Furthermore, one can use type theory as a terminating functional programming language.

By analogy with $\forall x : A.B$ it would be natural to add an operation $\exists x : A.B$ with the rules

$$\frac{\Gamma \vdash A \quad \Gamma, x : A \vdash B : \mathsf{U}_0}{\Gamma \vdash \exists x : A.B : \mathsf{U}_0}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B[M]}{\Gamma \vdash (M, N) : \exists x : A.B} \qquad \frac{\Gamma \vdash P : \exists x : A.B}{\Gamma \vdash P.1 : A} \qquad \frac{\Gamma \vdash P : \exists x : A.B}{\Gamma \vdash P.2 : B[P.1]}$$

A fundamental result, which plays a role in this paper, is that the addition of these rules is *contradictory*: it is possible to build a proof of $\bot$ (which is then automatically not normalisable) in this extended system [7].

It would be difficult to try and give an intuitive reason why the system with $\exists$ is inconsistent. Unfortunately the proof of inconsistency is not so intuitive. In this particular case, it is possible to encode a type theory with a type of all types, which is a well known case of inconsistent theory [7].

Since these rules are contradictory, we cannot use them to represent mathematics. It is possible however to define $\exists x : A.B : \mathsf{U}_0$ as

$$\forall P : \mathsf{U}_0.(\forall x : A.B \Rightarrow P) \Rightarrow P$$

Since $B : \mathsf{U}_1\ [x : A]$ we can also form $\Sigma x : A.B : \mathsf{U}_1$. We then have a map

$$(\Sigma x : A.B) \to \exists x : A.B$$

but in general there is no map in the other direction.

3 Representation of Bishop set theory in type theory

3.1 Bishop sets

Bishop [3] specified the notion of set by stating that a set has to be given by a description of how to build element of this set and by giving a binary relation of equality, which has to be an equivalence relation. A function from a set A to a set B is then given by an *operation*, which is compatible with the equality (*i.e.* two elements which are equal in A are mapped to two elements which are equal in B), and is described as "a finite routine f which assigns an element $f(a)$ of B to each given element a of A". This notion of routine is left informal but must "afford an explicit, finite, mechanical reduction of the procedure for constructing $f(a)$ to the procedure for constructing a." It is direct and natural to represent formally all these notions in our type theory.

A *Bishop set* is defined to be a type $A : \mathsf{U}_1$ together with an *equivalence relation* over A that is an element $R : A \to A \to \mathsf{U}_0$ with a proof of $equiv\ A\ R$, where

$$equiv : \Pi A : \mathsf{U}_1.(A \to A \to \mathsf{U}_0) \to \mathsf{U}_0$$

$$equiv\ A\ R = refl\ A\ R\ \wedge\ sym\ A\ R\ \wedge\ trans\ A\ R$$

$$refl\ A\ R = \forall x : A.R\ x\ x, \quad sym\ A\ R = \forall x\ y : A.\ R\ x\ y \Rightarrow R\ y\ x$$

$$trans\ A\ R = \forall x\ y\ z : A.\ R\ x\ y \wedge R\ y\ z \Rightarrow R\ x\ z$$

It is possible to represent the collection of all Bishop sets as the type

$$\Sigma A : \mathsf{U}_1.\Sigma R : A \to A \to \mathsf{U}_0.equiv\ A\ R$$

which is itself an element of type U_2.

If $X = (A, R, p)$ is a Bishop set, we write $|X| = X.1 = A$ its corresponding type and $=_X$ for its corresponding equivalence relation $R = X.2.1$. If there is no confusion, we shall say simply "set" for "Bishop set". If $Y = (B, S, q)$ is another set, then one can form the set Y^X of *functions* from X to Y by taking

$$|Y^X| = \Sigma f : |X| \to |Y|.\forall x_1\ x_2 : |X|.\ \ x_1 =_X x_2 \Rightarrow f\ x_1 =_Y f\ x_2$$

and $(f_1, p_1) =_{Y^X} (f_2, p_2)$ is the proposition $\Pi x : A.\ \ f_1\ x =_Y f_2\ x$.

Bishop sets form a category. One can ask how similar is this category to the category of sets in ZF. We analyse this in the next subsection.

3.2 Truth values, properties and subsets

An important set is the set of truth values Ω such that $|\Omega|$ is U_0 and $=_\Omega$ is $\Leftrightarrow$. A *property* on X is a function from X to Ω (in Bishop's sense).

If $P : A \to \mathsf{U}_0$ is a property on a set $X = (A, =_X, p)$, which means that $x_1 =_X x_2$ implies $P\ x_1 \Leftrightarrow P\ x_2$, it is possible to define the set $|Y| = \Sigma x : A.P\ x$ with the equality $(x_1, p_1) =_Y (x_2, p_2)$ iff $x_1 =_X x_2$. One can then check that the map $m : y \longmapsto y.1$ is a function from the set Y to the set X which is one-to-one: $y_1 =_Y y_2$ iff $m\ y_1 =_X m\ y_2$. Bishop defines a *subset* of X to be such a function $i : Z \to X$ which is one-to-one. We have just seen that any property P on X defines a subset $m : Y \to X$ of X, and it is natural to write $m : \{x : X \mid P\ x\} \to X$ for this subset. This corresponds to the comprehension axiom in systems such as ZF.

Conversely, given a subset Z with $|Z| = C$, $i : C \to A$ it is possible to define a property $P : A \to \mathsf{U}_0$ on X by taking $P\ x$ to be $\exists z : C.x =_X i\ z$. This property defines a subset $m : Y \to X$ with $Y = \{x : X \mid P\ x\}$. In ZF set theory the two subsets Y and Z are equivalent and it is possible to find a (unique) map $f : Y \to Z$ such that $i\ f = m$. This is *not* possible in our representation: given $x : A$ and a proof that $\exists z : C.x =_X i\ z$ there is no way in general to extract from this proof an element $z : C$ such that $x =_X i\ z$ holds. In general, we do *not* have the implication

$$(\forall x : A.\exists! z : C.R\ x\ z) \to \exists f : A \to C.\forall x : A.R\ x\ (f\ x) \qquad (*)$$

In set theory, this implication is achieved by reducing *functions* to *functional relations*. However, we want here to be able to use functions as *functional programs* for our representation of Kenzo. Since functional programs do not coincide with functional relations, it is natural that the implication $(*)$ is not valid.

From these remarks, one can see that the category of Bishop sets we have defined is *not* a topos. It thus differs in subtle way from the usual category of sets. We shall see similarly that in this setting the category of abelian groups is *not* an abelian category (but the category of finitely presented abelian groups is). However, and this is an important point, it is not an obstacle in representing Kenzo. (This can be expected since the goal of Kenzo is precisely to obtain functional programs, and not abstractly defined relations.)

3.3 Alternative representation

Following Curry-Howard, one can represent propositions as types in U_1. We don't need then to introduce the type U_0. Existential propositions are then represented using sigma types, and in this representation, there is a good correspondance between subsets and properties and the implication $(*)$ is valid. The problem there is that we don't have a *set* of truth values any more, since the type of propositions, U_1, is itself of type U_2. (So the category of sets do not form a topos either in this representation.)

We feel that our representation is closer to mathematical practice, and separates more clearly what is the computational part, at level U_1, and the specification part, at level U_0[4].

4 Category theory in type theory

We can represent the type of "all" Bishop sets, and this itself is of type U_2. It is possible similarly to represent the collection of all groups, the category of all sets, the category of all groups, and these are represented by types that are in U_2. (This corresponds to the notion of locally small categories.) One can then also consider the 2-category of all these categories, and this will be represented by a type in U_3.

In general, a locally small category will be represented by a *type* of objects $Obj : \mathsf{U}_2$ (for instance the type of Bishop sets). If $A, B : Obj$ we suppose given a *set* $Hom\ A\ B$ of morphisms. Thus, if $A, B : Obj$ we have $Hom\ A\ B : \mathsf{U}_1$ and we have an equality $f =_{Hom\ A\ B} g$ which is in U_0 for $f, g : Hom\ A\ B$. We introduce also the identity morphism, the composition operator, and the usual axioms of associativity and identity.

An important instance is provided by the category of abelian groups[5].

4.1 Properties of the category of abelian groups

In classical mathematics, an elegant axiomatisation of the category of abelian groups is provided by the notion of *abelian category*. What are the properties of the category of abelian groups represented in type theory?

[4] For the development of Kenzo however, both approaches seem possible.

[5] The main difference with the treatment in [9] is the following. We use the structure of universes to stratify the categories: locally small categories are represented with a type of object in U_2, 2-categories with a type of objects in U_3, ...

It is clear that this category is *preabelian*: it is preadditive, all hom-sets are abelian groups and the composition of morphisms is bilinear, it is additive since we can form finite direct sums and direct products, and finally, every morphism has both a kernel and a cokernel. This can also be quite directly checked formally.

So the category of abelian groups represented in type theory is preabelian. Is it abelian? Surprisingly it is *not* the case that every monomorphism and every epimorphism is normal (that is, a monomorphism for instance is not necessarily the kernel of a map). If we have a map $u : A \to B$ which is mono, that is $u\ x = 0 \to x = 0$ then, in usual set theory, this map is the kernel of the map $s : B \to B/Im\ u$ (which always makes sense since all the groups are abelian). This means that if we have a map $f : X \to B$ such that $s\ f = 0$ then there exists a unique map $g : X \to A$ such that $f = u\ g$. That $s\ f = 0$ means that for all $x \in X$ there exists $a \in A$ (unique) such that $f\ x = u\ a$. But it is not possible in our representation of Bishop sets to deduce from this the existence of a map $g : X \to A$ such that $f\ x = u\ (g\ x)$. One would need for this the implication

$$(\forall x : A.\exists! z : C.R\ x\ z) \to \exists f : A \to C.\forall x : A.R\ x\ (f\ x) \qquad (*)$$

that, as we have seen, does not hold in general.

In any case, we have chosen to axiomatise the algebraic reasoning justifying Kenzo at the level of *preabelian* category and not at the level of *abelian* category. We believe that actually this reflects better the reasonings done in [15][6].

4.2 Preabelian category

We represent the notion of general preabelian category in type theory in the following way. (All these axioms are instantiated by the category of abelian groups.)

First we have a type of objects $Obj : \mathsf{U}_2$. We have to use the type U_2 since it is the type of the collection of all abelian groups, represented as a sigma type. For any two objects $A, B : Obj$ we have an abelian group of morpshims $Hom\ A\ B$. Thus $Hom\ A\ B : \mathsf{U}_1$ and we have an equality on $Hom\ A\ B$ and a group operation $f + g : Hom\ A\ B$ for $f, g : Hom\ A\ B$ with a zero element $0 : Hom\ A\ B$. We have a composition operation $gf : Hom\ A\ C$ for $g : Hom\ B\ C$ and $f : Hom\ A\ B$. We require the equations $(f + g)h = fh + gh$ and $h(f + g) = hf + hg$.

There is a zero object $0 : Obj$ such that $f = 0$ if $f : Hom\ A\ 0$ or $f : Hom\ 0\ A$. We have biproducts, and there is an operation $(+) : Obj \to Obj \to Obj$ with morphisms $i : Hom\ A\ (A + B)$, $j : Hom\ B\ (A + B)$ and $p : Hom\ (A + B)\ A$ and $q : Hom\ (A + B)\ B$ with equations $pi = 1,\ qj = 1,\ pj = 0,\ qi = 0,\ ip + jq = 1$.

For stating the existence of the kernel, it is convenient to use the telescope notation [4]

$$(Ker, inj, pinj) : \Pi A\ B : Obj.\Pi f : Hom\ A\ B.\ (K : Obj,\ i : Hom\ K\ A,\ f\ i = 0)$$

We require also the universal condition

$$(univ, puniv) : \Pi X : Obj.\Pi u : Hom\ X\ A.\ f\ u = 0 \to (v : Hom\ X\ (Ker\ f),\ u = (inj\ f)\ v)$$

and the unicity condition

$$\Pi X : Obj.\Pi u : Hom\ X\ A.\Pi p : f\ u = 0.\Pi v : Hom\ X\ (Ker\ f).\ u = (inj\ f)\ v \to v = univ\ f\ u\ p$$

We state the existence of cokernel in a dual way.

We can define in this way a sigma type $PreAb : \mathsf{U}_3$ which represents the collection of all prebalian categories. In particular one can define an element of type $PreAb$ which is the category of all abelian groups. One could also define the notion of additive functors between two elements of type $PreAb$.

[6] Let us consider as an example the long exact sequence of a short exact sequence (section 2.6 of [15]). This is something that only can be done in an *abelian* category. However, the Kenzo version of this notion requires a further hypothesis. The exactness property of the short exact sequence must be *effective* (definition 80 of [15]). With this extra hypothesis, the reasoning can then be represented at the level of preabelian category.

4.3 Implicit arguments

Besides dependent types, we use an important notational facility: *implicit arguments*. We don't need to give explicitly the arguments that can be inferred from the context. For instance the composition operator is of type

$$\Pi\ A\ B\ C : Obj.\ Hom\ A\ B \to Hom\ B\ C \to Hom\ A\ C$$

and expects 5 arguments. Since the 3 first arguments can be inferred from the last 2 arguments, one needs only to give the last 2 arguments and can write the composition (almost) as usual. In this way, we can write gf instead of $comp\ A\ B\ C\ f\ g$.

5 Formalisation of Kenzo

The main idea is to represent the reasonings done in [15] in an arbitrary preabelian category.

5.1 A test example

We have tested this approach on the introductory example Lemma 3.3.1 of [1]. As stated in [1], this example seems to contain the most interesting problems that have been found in the formalisation of proofs in Kenzo. It requires reasoning with homomorphisms and endomorphisms as if they were elements of certain algebraic structures, but also dealing with their functional definition; furthermore the domain conditions on the source or the target of the homomorphisms also are important. We try here to approach these issues by a formalisation at the level of category theory. The domain and codomain are explicit, but can be hidden since they can be inferred from the context.

We work in an arbitrary preabelian category. We suppose that we have $h, d : G \to G$ such that $dd = hh = 0$ and $hdh = h$. We define $p = dh + hd$. We consider then the inclusion $i : K \to G$ where $K = Ker\ p$.

Proposition 1. *We have*

$$pp = p,\ ph = hp = h,\ pd = dp,\ hi = pi = 0$$

It follows that $pdi = dpi = 0$. *Hence there exists* $d_1 : K \to K$ *such that* $id_1 = di$. *We then have* $d_1 d_1 = 0$. *Since* $p(1-p) = 0$ *there exists* $j : G \to K$ *such that* $ij = 1 - p$ *and we have* $jh = 0,\ jd = d_1 j$.

Proof. The equality $pp = p$ is proved by computation

$$(hd + dh)(hd + dh) = hdhd + hddh + dhhd + dhdh = hd + 0 + 0 + dh = hd + dh = p$$

Similarly we check $hp = hhd + hdh = 0 + h = h$, $ph = hdh + dhh = h + 0 = h$ and $dp = ddh + dhd = dhd = dhd + hdd = pd$. Since i is mono, $id_1 d_1 = did_1 = ddi = 0$ implies that $d_1 d_1 = 0$.

(The proposition can be stated as the fact that (j, i, h) is a *reduction* from G, d to K, d_1, as defined below.)

5.2 Use of category theory

This "pointfree" style, which requires to represent formally some basic notion of category theory, can be compared to the formalisation in [1].

The statement of Proposition 1 is not exactly the same as the one of Lemma 3.3.1 of [1] or even the corresponding informal statement in [15]. The formal representation in [1] allows to consider for instance $1 - p$ both as a map of type $G \to G$ and of type $G \to K$. We do not allow this, and have to distinguish between $j : G \to K$ and $1 - p : G \to G$ such that $ij = 1 - p$. We do not think however that this is a problem in practice.

The point is that the proof is essentially equational, like in a non commutative ring, but with an addition and a multiplication operations that are not always defined (the arity should be compatible). We can furthermore represent it as it is in type theory (see the appendix). We believe that this formalisation is close to a precise informal mathematical reasoning.

5.3 Use of dependent types

The extension of higher-order logic to a type system with universes is natural to represent category theory. It seems also *necessary* if one wants to have a system in which one can state general properties of an arbitrary category, and then instantiate it on concrete categories.

Dependent types are also used to facilitate *modular* reasoning. We can state a property about an arbitrary preabelian category, and then instantiate it to the category of abelian groups.

Finally, in this representation, all the terms can directly be seen as functional programs.

5.4 Refinement of the test example

In an arbitrary preabelian category, we define a *differential object* to be a pair $G, d : G \to G$ with $dd = 0$. If $G, d : G \to G$ is a differential object, we define the homology $H(G, d)$ of G, d as follows. We consider $m : Ker\ d \to G$. Since $dd = 0$ there exists a map $d' : G \to Ker\ d$ such that $md' = d$. This map has a cokernel $s : Ker\ d \to H(G, d)$.

If $G_1, d_1 : G_1 \to G_1$ is another differential object, and $f : G \to G_1$ satisfies $fd = d_1 f$, we can build a map $H(f) : H(G, d) \to H(G_1, d_1)$ characterised by the condition $s_1 d'_1 fm = H(f)s$.

A *reduction* from G, d to G_1, d_1 is a triple f, g, h such that $f : G \to G_1,\ g : G_1 \to G,\ h : G \to G$ such that $d_1 f = fd,\ gd_1 = dg,\ fg = 1,\ gf = 1 - hd - dh,\ hh = fh = hg = 0$.

Proposition 2. *If f, g, h is a reduction from G, d to G_1, d_1 then $H(f),\ H(g)$ define an isomorphism between $H(G, d)$ and $H(G_1, d_1)$, that is $H(g)H(f) = 1$ and $H(f)H(g) = 1$.*

6 Main remaining steps

We believe that the notion of preabelian category gives the right axiomatic level to formally represent what is going on in Kenzo. The next natural step is to represent the previous notion of equivalence and homology group for *chain-complexes*. This is naturally represented in a system with dependent types and we don't expect essential problems here. We hope then to be able to represent formally the Basic Pertubation Lemma as it is formulated in [15] (but for an arbitrary preabelian category). We should then, using [15] and the Pertubation Lemma as an essential tool, develop a library of programs to reduce a chain-complex to a chain-complex of finitely presented modules, for which one can compute explicitly the homology group.

The computationally challenging part is to represent the notion of *finitely presented* abelian group, and the main algorithms on these groups: effective computation of the kernel, cokernel via Smith reduction. This would involve computations on matrices of integers, and may be done by formalising Chapter V 1 of [14]. Using these algorithms one can then compute a canonical representation of the homology groups of a chain-complex of finitely presented modules. This should be a perfect test for the new representation of integers in type theory [16].

Appendix: formal representation in Coq

We have represented formally the notion of preabelian category, and checked that abelian groups form a preabelian category. We have then formulated and proved Lemma 3.3.1 of [1] in an arbitrary prebalian category.

Sum-up of the syntactic facilities

Coq provides a lot of syntactic sugar to ease the work of the user. Here is a few words about these:

1. Record types: the syntax `Record` *name* `:` *univ* `:=` *c* `{` *field* `:` *type* [;...] `}` allows to declare dependent tuple-types (*i.e.* n-ary Σ types) with named field. This is actually no extension of the theory shown in Sections 1 and 2. Internally, this syntax declares a regular Σ-type that

goes in universe *univ*, and name it *name*. It also declares a *constructor i.e.* a term of type Πf:*type*$\ldots$.*name* this term has a fundamental meaning in Coq, but in a first approximation, one can see it as a tool for tactic-based proofs. Finally it declares a function *field* of type *name* $\rightarrow$ *type* for each field declaration (*field* `:` *type*). It is rather useful to use this notation when building very large tuples like the definition of a preabelian category.

2. Implicit coercions: in Coq, one can declare a function f as an implicit coercion between two *classes* of types, a class being basically an identifiable type construction. If f is a coercion from type t to type u, then f is automatically inserted whenever a term of type t is given where a term of type u is expected. This allows to consider a category as a type: for a category `C` there is a type of its object `dom C`. If we declare `dom` as a coercion, then we can "abuse notations" and write simply `C` instead of `dom C`.
3. Implicit coercions in record definition: as an additional notation we can define fields of a record as implicit coercions during the definition of the record with the syntax —*field* `:>` *type* instead of the usual —*field* `:` *type*. An intuitive way to read this syntactic construction would be to consider the new record type as an extension of *type*. For instance :
   `Record preabelian_category : Type := mk_preabcat { preab_cat :> category; ... }`
   reads "a preabelian category is a category with `...`".

Preabelian category

Here are additional notes to read this formalisation :

- The composition of `f` and `g` is written `f!g` (instead of `gf`).
- In Coq, Πx:A.B and $\forall x$:A.B are both written `forall x:A,B`. The design choices have led to use the dot as the end-of-line symbol, so a coma was use in the `forall` construction instead.
- The definition of the type `category` is not included, but it is worth noticing that both the domain function `dom` and the hom-set `hom` function have been declared as coercions. Thus we can write `C` instead of `dom C` and `C X Y` instead of `hom C X Y`. As an example, `forall (X Y:C) (f:C X Y), f == f` reads "for all objects `X` and `Y` of the category `C` and for all arrow `f` in the hom-set from `X` to `Y`, `f` is equal to itself"

```
Require Export category.

Record preabelian_category : Type := mk_preabcat {
  preab_cat :> category;
  zero : preab_cat;
  zero_is_zero : zero_object zero;
  zerom : forall (X Y:preab_cat), preab_cat X Y;
  zerom_is_zero : forall (X Y:preab_cat), zero_morphism zero (zerom X Y);
  hom_plus : forall (X Y:preab_cat) (f g:preab_cat X Y), preab_cat X Y;
  hom_plus_morphism : forall (X Y:preab_cat) (f1 f2 g1 g2:preab_cat X Y),
                            f1==f2 -> g1==g2 ->
                            hom_plus  f1 g1 == hom_plus  f2 g2;
  hom_plus_assoc : forall (X Y:preab_cat) (f g h:preab_cat X Y),
                            hom_plus  f (hom_plus  g h ) ==
                            hom_plus  (hom_plus  f g) h;
  hom_plus_comm : forall (X Y:preab_cat) (f g:preab_cat X Y),
                            hom_plus  f g == hom_plus  g f;
  hom_plus_zero_l : forall (X Y:preab_cat) (f:preab_cat X Y),
                            hom_plus  zerom  f == f;
  hom_plus_zero_r : forall (X Y:preab_cat) (f:preab_cat X Y),
                            hom_plus  f zerom == f;
  hom_minus : forall (X Y:preab_cat) (f:preab_cat X Y), preab_cat X Y;
  hom_minus_morphism : forall (X Y:preab_cat) (f1 f2:preab_cat X Y),
```

```
                          f1 == f2 -> hom_minus  f1 == hom_minus  f2;
hom_minus_is_minus_l : forall (X Y:preab_cat) (f:preab_cat X Y),
                          (hom_plus  f (hom_minus  f)) == zerom ;
hom_minus_is_minus_r : forall (X Y:preab_cat) (f:preab_cat X Y),
                          (hom_plus  (hom_minus  f) f) == zerom ;
comp_plus_linear : forall (X Y Z:preab_cat) (f1 f2:preab_cat X Y)
                         (g:preab_cat Y Z),
                          (hom_plus  f1 f2)!g ==
                          hom_plus  (f1!g) (f2!g);
comp_plus_colinear : forall (X Y Z:preab_cat) (f:preab_cat X Y)
                           (g1 g2:preab_cat Y Z),
                          f!(hom_plus  g1 g2) ==
                          hom_plus  (f!g1) (f!g2);
comp_minus_linear : forall (X Y Z:preab_cat) (f:preab_cat X Y)
                          (g:preab_cat Y Z),
                          (hom_minus  f)!g == hom_minus  (f!g);
comp_minus_colinear : forall (X Y Z:preab_cat) (f:preab_cat X Y)
                            (g:preab_cat Y Z),
                          f!(hom_minus  g) == hom_minus  (f!g);
biproduct :
  forall (A B:preab_cat),
   {A_B:preab_cat &
    {pA:preab_cat A_B A &
     {pB:preab_cat A_B B &
      {iA:preab_cat A A_B &
       {iB:preab_cat B A_B |
        hom_plus  (pA!iA) (pB!iB) == id /\
        iA!pA == id /\ iB!pB == id /\
        iA!pB == zerom  /\ iB!pA == zerom }}}}}
ker_obj : forall (X Y : preab_cat) (f:preab_cat X Y), preab_cat
ker_arr : forall (X Y : preab_cat) (f:preab_cat X Y),
                        preab_cat (ker_obj  f) X;
ker_univ_arr : forall (H X Y : preab_cat) (f:preab_cat X Y)
                      (h:preab_cat H X) (p: h!f == zerom ),
                        preab_cat H (ker_obj  f);
ker_univ_com : forall (H X Y : preab_cat) (f:preab_cat X Y)
                      (h:preab_cat H X) (p: h!f == zerom ),
                        h == (ker_univ_arr  f h p)!(ker_arr  f);
ker_univ_uniq : forall (X Y:preab_cat) (f:preab_cat X Y),
                       (h:preab_cat H X) (p: h!f == zerom ),
                        (i:preab_cat H (ker_obj  f)),
                        (q: h == i!(ker_arr  f)),
                        i == ker_univ_arr  f h p;
coker_obj : forall (X Y : preab_cat) (f:preab_cat X Y), preab_cat
coker_arr : forall (X Y : preab_cat) (f:preab_cat X Y),
                        preab_cat X (coker_obj  f);
coker_univ_arr : forall (H X Y : preab_cat) (f:preab_cat X Y)
                        (h:preab_cat Y H) (p: f!h == zerom ),
                        preab_cat (coker_obj  f) H;
coker_univ_com : forall (H X Y : preab_cat) (f:preab_cat X Y)
                        (h:preab_cat X H) (p: f!h == zerom ),
                        h == (coker_arr  f)!(coker_univ_arr  f h p);
coker_univ_uniq : forall (X Y:preab_cat) (f:preab_cat X Y),
                        (h:preab_cat Y H) (p: f!h == zerom ),
```

```
                          (i:preab_cat (coker_obj  f) H),
                          (q: h == (coker_arr  f)!i),
                          i == coker_univ_arr f h p;
}.

Notation "0" := (zerom) : preabelian_category_scope.
Notation "f + g" := (hom_plus f g) : preabelian_category_scope.
Notation "~ f" := (hom_minus f) : preabelian_category_scope.
Notation "f - g" := (f+~g)%preab : preabelian_category_scope.
```

The test example in type theory

We use implicit coercions to be able to write write both `C` for the type `obj C` of objects of `C` and `C A B` for the type `hom C A B` of morphisms from `A` to `B`.

```
Variable C:preabelian_category.

Variable G:C.
Variable h d:C G G.

Variable (dd_zero: d!d == 0) (hh_zero : h!h == 0) (hdh_h : h!d!h == h).

Definition p := d!h+h!d.

Lemma ph_h : p!h == h.

Lemma uv_v_implies_id_minus_u_v_zero :
 forall u v:C G G, u!v == v -> (id-u)!v == 0.

Lemma id_minus_p_h_zero : (id-p)!h == 0.

Lemma hp_h : h!p == h.

Lemma vu_v_implies_v_id_minus_u_zero :
 forall u v:C G G, v!u == v -> v!(id-u) == 0.

Lemma h_id_minus_p_zero : h!(id-p) == 0.

Lemma pp_p : p!p == p.

Lemma id_minus_p_p_zero : (id - p)!p == 0.

Lemma p_id_minus_p_zero : p!(id - p) == 0.

Definition K := ker_obj p.
Definition i : C K G := ker_arr p.

Lemma ip_zero : i!p == 0.

Lemma ih_zero : i!h == 0.

Definition j : C G K := ker_univ_arr p (id - p) id_minus_p_p_zero.
```

```
Lemma ji_id_minus_p : j!i == id - p.

Lemma hj_zero : h!j == 0.
```

References

1. J.M. Aransay. Razonamiento mecanizado en álgebra homológica. PhD thesis, 2006.
2. M. Artin, A. Grothendieck, J,-L. Verdier. Univers. Séminaire de Géométrie Algébrique du Bois Marie - 1963-64 - Théorie des topos et cohomologie étale des schémas - (SGA 4) - vol. 1, LNM 269, 185-217, Berlin; New York: Springer-Verlag.
3. E. Bishop. *Foundations of Constructive Analysis.* New York: McGraw-Hill 1967.
4. N.G. de Bruijn. Telescopic mappings in typed lambda calculus. Inform. and Comput. 91 (1991), no. 2, 189–204.
5. M. Barakat and D. Robertz, homalg: First steps to an abstract package for homological algebra. Proceedings of the X meeting on computational algebra and its applications (EACA 2006), Sevilla (Spain), 2006, pp. 29-32.
6. LogiCal project. The Coq Proof Assistant. http://coq.inria.fr/V8.1/refman/index.html, 2007.
7. Th. Coquand. Metamathematical investigations of a calculus of constructions. Rapport de recherche de l'INRIA, 1989.
8. B. Gregoire and X. Leroy. A Compiled Implementation of Strong Reduction. International Conference on Functional Programming, 2002.
9. G. Huet and A. Saibi. Constructive category theory. Proof, language, and interaction, 239–275, Found. Comput. Ser., MIT Press, Cambridge, MA, 2000
10. P. Landin. The mechanical evaluation of expressions. Comput. J., 6, 308-320, 1964.
11. P.Martin-Löf. An intuitionistic theory of types. in *Twenty-five years of constructive type theory* (Venice, 1995), 127–172, Oxford Logic Guides, 36, Oxford Univ. Press, New York, 1998.
12. P.-A. Melliès and B. Werner. A Generic Normalization Proof for Pure Type Systems. in: TYPES'96, C. Paulin-Mohring, E. Gimenez (rd.), LNCS, Springer-Verlag, 1997.
13. A. Miquel and B. Werner. The Not So Simple Proof-Irrelevant Model of CC. in: Types for Proofs and Programs (TYPES'02). 2003.
14. R. Mines, F. Richman and W. Ruitenburg. *A Course in Constructive Algebra.* Springer-Verlag, 1988.
15. J. Rubio and F. Sergereart. *Constructive Homological Algebra and Applications.* 2006 Genove Summer School. Available at http://www-fourier.ujf-grenoble.fr/∼sergerar/Papers/
16. A. Spiwack. Ajouter des entiers machine à Coq. Master thesis, http://arnaud.spiwack.free.fr/